Verifying Mutual Exclusion with TLA$^+$

A thesis presented in partial fulfillment for the degree of

Bachelor of Science

Peter Lindner

Supervisor: Assoc. Prof. Dr. Ana Sokolova

Paris-Lodron University Salzburg

Department of Computer Science
Faculty of Digital and Analytical Sciences
Salzburg, Austria

August 2024

Declaration

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Salzburg, August 16, 2024

Peter Lindner

Contents

Introduction **-1**

1 Preliminaries **0**

1.1 Temporal Logic of Actions . . . 0

1.1.1 Syntax . . . 0

1.1.2 Semantics . . . 1

1.1.3 System Properties . . . 9

1.2 TLA$^+$ . . . 11

1.2.1 Modules . . . 11

1.2.2 Additional Notions . . . 12

1.2.3 Toolbox . . . 15

1.3 Model Checking . . . 16

1.3.1 State Space Explosion . . . 17

1.3.2 TLC . . . 17

2 Verifying Mutual Exclusion **20**

2.1 Methodology . . . 20

2.1.1 Model . . . 20

2.1.2 Properties . . . 23

2.1.3 Verification . . . 23

2.2 Szymanski's Flag Algorithm . . . 26

2.2.1 Specification . . . 28

2.2.2 Results . . . 29

2.3 Szymanski's Flag Algorithm using Bits . . . 29

2.3.1 Specification . . . 31

2.3.2 Results . . . 31

3 Conclusion **34**

A TLA$^+$ Specifications **37**

A.1 Szymanski's Flag Algorithm . . . 37

A.2 Szymanski's Flag Algorithm using Bits . . . 41

B TLC Configurations **46**

Introduction

In recent decades, software systems found their way into almost every domain, including safety-critical systems. History has shown that failures in such software systems can result in fatal consequences and may cause huge financial losses or even threatening human lives. A notable example of such a failure is the explosion of the Ariane 5 rocket 40 seconds into its V88 flight, as stated in [1]. The explosion was caused by an overflow during a simple type conversion from a 64-bit floating point number to a 16-bit integer, leading to an unhandled exception and ultimately causing the computer to fail entirely. While in this case only financial damage was incurred, the Therac-25 incident reported on in [2] resulted in an even more tragic outcome. Therac-25 was a radio therapy machine that suffered from software errors in its control unit. These errors caused the machine to emit too much radiation, leading to the deaths of six patients during the 1980s. Incidents like these raise an important question:

How does one prevent errors of this kind?

In [1], the authors mention four different methods to ensure the functionality of such complex systems according to their specifications: simulation, testing, model checking and deductive verification. While simulation and testing offer cost-efficient ways to assess system outputs based on certain inputs, most often it is infeasible to take every possible input into account. Hence, for safety-critical systems, a more sophisticated way of reasoning about correctness is required.

In this thesis, we will focus on model checking with the aim to verify two variants of a mutual exclusion algorithm using the TLC model checker. Since TLC is designed to verify TLA$^+$ specifications, we first have to lay the necessary preliminaries. Therefore, we start by giving an introduction to the Temporal Logic of Actions (TLA), which forms the foundation of the TLA$^+$ specification language. TLA is a variant of linear-time temporal logic and has been developed by Lamport [3, 4] to reason about concurrent algorithms by using the same logic for both specification and implementation. Algorithms are defined with a logical expression, leading to a description that is much more concise than if expressed in a programming language. Based on this introduction we will discuss TLA$^+$ and its toolbox, including the TLC model checker.

Over the years, TLA$^+$ has been successfully employed in the industry by companies such as Amazon [5, 6], Intel [7] or Microsoft [8] and has been utilized in the development of reliable real-time operating systems [9]. Recently, the Linux Foundation announced the establishment of the TLA$^+$ foundation, with Amazon, Microsoft, and Oracle as its founding members [10].

1. Preliminaries

In this chapter, we will cover all the preliminary information necessary to comprehend the methods used in Chapter 2. Initially, we introduce the Temporal Logic of Actions, followed by a discussion on TLA$^+$ and its toolbox. Afterwards, we will give a brief overview of model checking and TLC, the model checker for TLA$^+$ specifications.

1.1 Temporal Logic of Actions

The following introduction to the temporal logic of actions (TLA) is based on [3, 4]. Here, we assume familiarity with first-order logic, as its constructs are used throughout this introduction. We first introduce the syntax of TLA, followed by its semantics, while omitting details not needed for our purpose. For a complete introduction, we refer the reader to [3].

1.1.1 Syntax

Let V_S be a finite set of *state variable* names and V_R be a finite set of *rigid variable* names, where $V_S \cap V_R = \emptyset$. We define the set of primed state variables as $V'_S \coloneqq \{v' \mid v \in V_S\}$. Furthermore, let L_F denote the set of all function symbols and L_P the set of all predicate symbols of given arity, with $L_F \cap L_P = \emptyset$. For convenience, we define the sets V and L as

$$V \coloneqq V_S \cup V'_S \cup V_R \qquad\qquad L \coloneqq L_F \cup L_P$$

where $V \cap L = \emptyset$. Note that we define these sets to be finite due to the nature of their application.

Definition 1.1 (Formula)**.** The syntax of a *formula* is given by

$$\phi \coloneqq \; P \mid \phi \wedge \phi \mid \neg\phi \mid \Box\phi \mid \Box[a]_f \mid \exists v : \phi \mid \exists_R v_r : \phi$$

where

- P is a *predicate* symbol over $V_S \cup V_R$, called a *state predicate*
- $\Box$ is the *always*-operator
- a is a *predicate* symbol over $V \cup L$, called an *action*
- f is an *expression*, called a *state function*
- $v \in V_S$ is a *variable* name
- $v_r \in V_R$ is a *rigid variable* name
- $\exists$ is an *existential quantifier* over a $v \in V_S$
- $\exists_R$ is an *existential quantifier* over a $v_r \in V_R$ ▶

1.1.2 Semantics

To give this abstract syntax meaning, we now define the semantics of TLA. In the following, we consider all function and predicate symbols to be pre-instantiated. Given that the sets of variables V_S and V_R are finite and the functions and predicates are pre-instantiated, we omit specifying the variables they operate on, as these are implicitly understood. Note that an expression in TLA is treated equivalently to a term in first-order logic, and its evaluation follows the standard semantics of first-order logic. We will not introduce the semantics of the $\exists$-quantifier on state variables and the $\exists_R$-quantifier on rigid variables, as they are not relevant in this thesis. In general, for every syntactic object F, we introduce its semantics $[\![F]\!]$, except for *values* and *states*, which are purely semantic concepts.

Definition 1.2 (Values)**.** Let Val be a set of *values* used for reasoning about an algorithm. We note that $\mathit{true}, \mathit{false} \notin \mathrm{Val}$. ▶

Definition 1.3 (States)**.** A function $s : V_S \to \mathrm{Val}$ is a *state*, where $s(x) \in \mathrm{Val}$ is the corresponding value of a variable $x \in V_S$ in the given state s. We write $s[\![x]\!]$ for $s(x)$ and refer to the set of all states as St. ▶

Definition 1.4 (State function)**.** Let $V_S = \{v_1, v_2, \ldots, v_n\}$ be the set of state variables. The semantics of a state function f is given by a function $[\![f]\!] : \mathrm{St} \to \mathrm{Val}$ and we write

$$s[\![f]\!] \text{ for } [\![f]\!]\,[s[\![v_1]\!]\,/\,v_1,\ \ldots,\ s[\![v_n]\!]\,/\,v_n].$$

Therefore, we substitute every variable v_i by its corresponding value $s[\![v_i]\!]$ in a given state s, thus mapping from the state s to the value $s[\![f]\!] \in \mathrm{Val}$, obtained by evaluating the expression with the values in a state s. Note that rigid variables do not get substituted, unlike state variables. ▶

We consider the following example:

Example 1.1. Let f be a state function given by

$$f \coloneqq x + 1,$$

where $x \in V_S$, and let s be a state with $s[\![x]\!] = 1$. Thus, we get

$$s[\![f]\!] = s[\![x + 1]\!] = (s[\![x]\!] + 1) = (1 + 1) = 2.$$

This calculation demonstrates that in the state s, where $s[\![x]\!] = 1$, evaluating the state function f results in the value 2. Note that "+" is the pre-instantiated binary function symbol whose interpretation is addition of natural numbers.

Definition 1.5 (State predicate)**.** Let $V_S = \{v_1, v_2, \ldots, v_n\}$ be the set of state variables. For a predicate P over $V_S \cup V_R$, its semantics $[\![P]\!]$ is a state predicate (or simply predicate), i.e., $[\![P]\!] : \mathrm{St} \to \{\mathit{true}, \mathit{false}\}$ and we write

$$s[\![P]\!] \text{ for } [\![P]\!]\,[s[\![v_1]\!] \,/\, v_1,\, \ldots,\, s[\![v_n]\!] \,/\, v_n].$$

Again, in this approach, we substitute every variable v_i with its corresponding value $s[\![v_i]\!]$ in a given state s, thereby mapping the state s to either *true* or *false*, depending on the values within this state s. ▶

To further clarify this concept, let us examine the following example:

Example 1.2. Let P be a predicate given by

$$P \coloneqq (x + y \geq 3) \wedge x \in \mathbb{N},$$

where $x, y \in V_S$. Furthermore, let s be a state with $s[\![x]\!] = 1$ and $s[\![y]\!] = 2$. Hence, we see that

$$\begin{aligned} s[\![P]\!] &\equiv s[\![(x + y \geq 3) \wedge x \in \mathbb{N}]\!] \\ &\equiv ((s[\![x]\!] + s[\![y]\!] \geq 3) \wedge s[\![x]\!] \in \mathbb{N}) \\ &\equiv ((1 + 2 \geq 3) \wedge 1 \in \mathbb{N}) \\ &\equiv ((3 \geq 3) \wedge 1 \in \mathbb{N}) \\ &\equiv \mathit{true}. \end{aligned}$$

We extend the previous definition of a state predicate to include the notion of the *satisfaction relation*.

Definition 1.6 (State satisfies a predicate)**.** Let s be a state and P be a predicate. We say that s satisfies P iff $s[\![P]\!] \equiv \mathit{true}$, written as $s \vDash P$. ▶

Therefore, we note that in Example 1.2 the given state s satisfies the predicate P, written as $s \vDash P$.
Now in order to reason about atomic operations of concurrent programs, we introduce *actions*.

Definition 1.7 (Action)**.** Let $s, t \in \mathrm{St}$ be states. The semantics of a predicate a over $V \cup L$ is given by an *action*, characterized by a function mapping from a pair of states (s, t), the "old" state s and the "new" state t, to *true* or *false*. Hereby, the unprimed variables refer to s and the primed ones to t. Again, note that rigid variables do not get substituted, unlike state variables. We have $[\![a]\!] : \mathrm{St} \times \mathrm{St} \to \{\mathit{true}, \mathit{false}\}$ and denote

$$\begin{aligned} s[\![a]\!]t \text{ for } [\![a]\!](s, t), \text{ i.e., } [\![a]\!]\,[&s[\![v_1]\!] \,/\, v_1,\, \ldots,\, s[\![v_n]\!] \,/\, v_n, \\ &t[\![v_1]\!] \,/\, v_1',\, \ldots,\, t[\![v_n]\!] \,/\, v_n'], \end{aligned}$$

where $v_1, \ldots, v_n \in V_S$ and $v_1', \ldots, v_n' \in V_S'$. Thus, to evaluate an action, we substitute both the unprimed and primed variables v_i, v_i' with their corresponding value in the given states s and t, respectively. ▶

We consider the following example for an action:

Example 1.3. Let s, t be states with $s[\![x]\!] = 1, t[\![x]\!] = 3$, and let a be an action defined as

$$a \coloneqq x' = x + 2.$$

Hence, we get

$$\begin{aligned} s[\![a]\!]t &\equiv s[\![x' = x + 2]\!]t \\ &\equiv (t[\![x]\!] = s[\![x]\!] + 2) \\ &\equiv (3 = 1 + 2) \\ &\equiv \mathit{true}. \end{aligned}$$

To further simplify reasoning about such actions, we introduce the notion of *steps.*

Definition 1.8 (a-step)**.** Let a be an action and let $s, t \in \mathrm{St}$ be states. We call the pair (s, t) an a-step iff $s[\![a]\!]t \equiv \mathit{true}$, written as $s \xrightarrow{a} t$ or $(s, t) \models a$. ▶

In terms of concurrent programs, this means that executing a in a state s can produce state t. Hence, referring back to Example 1.3, we can describe $s[\![a]\!]t$ as an a-step. Since we introduced state predicates, we can consider any state predicate P to be an action without primed variables. Thus, for two states $s, t \in \mathrm{St}$

$$s[\![P]\!]t \equiv s[\![P]\!]$$

holds and $s[\![P]\!]t$ is a P step iff $s[\![P]\!] \equiv \mathit{true}$. Symmetrically, if we exchange all variables of P with its primed correspondents following

$$P' \coloneqq P\,[v_1' \,/\, v_1,\, \ldots,\, v_n' \,/\, v_n],$$

where $v_1, \ldots, v_n \in V_S$, we obtain P', and we conclude

$$s[\![P']\!]t \equiv t[\![P']\!].$$

Definition 1.9 (Validity of an action)**.** Let a be an action and $s, t \in \mathrm{St}$ be states. We define the *validity* of a, denoted by $\models a$, as follows:

$$\models a \coloneqq \forall s, t \in \mathrm{St}.\ s[\![a]\!]t. \qquad ▶$$

Hence, an action a is valid iff, for every pair of states, it is possible to perform an a-step. Considering a state predicate P as an action, the validity of P can be similarly defined:

$$\models P \coloneqq \forall s \in \mathrm{St}.\ s[\![P]\!].$$

To enhance the expressiveness of TLA, we introduce the concept of *rigid variables.*

Definition 1.10 (Rigid variables)**.** A *rigid* variable $v_r \in V_R$ is a variable, whose value never changes. It differs from a constant, as we do not know which actual value it holds. ▶

An example of such a rigid variable could be the number of processes of a concurrent system.

Definition 1.11 (Constant expression)**.** A *constant expression* is an expression that consists of constants and rigid variables. ▶

While arguing about programs it is useful to distinguish if it is possible to make an a-step starting from a certain state s. This leads to the definition of an action a being *enabled*.

Definition 1.12 (Action is enabled)**.** Let a be an action. We define the semantics of the predicate $\mathit{enabled}(a)$ as

$$s[\![\mathit{enabled}(a)]\!] \coloneqq \exists t \in \mathrm{St}.\ s[\![a]\!]t.$$ ▶

We can now extend TLA by incorporating the temporal aspect, giving the semantics of (temporal) *formulas.* These formulas are evaluated based on *behaviors.*

Definition 1.13 (Behavior)**.** A *behavior* $\sigma \in \mathrm{St}^\infty$ is an infinite sequence of states, e.g. $\sigma = \langle s_0, s_1, s_2, \dots \rangle$, where $s_0, s_1, s_2, \dots \in \mathrm{St}$ and St^∞ denotes the set of all behaviors. This concept represents the continuous progression of state in a system. Informally, when referring to a state s_n in a behavior, it corresponds to the n-th moment in time. Furthermore, we define St^* as the set of *partial* behaviors. An element $\pi \in \mathrm{St}^*$ is a finite sequence of states. For convenience, we write:

- $\sigma(i)$ for the i-th state in σ,
- σ_i for the behavior which is the prefix up to the i-th state from σ (including s_i),
- σ^i for the behavior which is the suffix of σ starting from $\sigma(i)$ and
- $\sigma\pi$ for the concatenation of two, possibly partial, behaviors. ▶

Note that a behavior can be any sequence of states and does not necessarily have to be possible within a system. We later discuss *possible* behaviors, which are those that satisfy the specification of a system given by a formula ϕ.

Definition 1.14 (Predicates on behaviors)**.** Let P be a predicate and $\sigma \in \mathrm{St}^\infty$ be a behavior given by $\sigma = \langle s_0, s_1, s_2, \dots \rangle$. We write

$$\sigma[\![P]\!] \text{ for } s_0[\![P]\!].$$

Hence, a predicate is evaluated by the first state of a behavior. We say that a behavior *satisfies* a predicate, written as $\sigma \vDash P$, iff $\sigma[\![P]\!] \equiv \mathit{true}$. ▶

Similarly to predicates on behaviors, we now study how actions are evaluated on behaviors.

Definition 1.15 (Actions on behaviors)**.** Let a be an action and $\sigma \in \mathrm{St}^\infty$ be a behavior given by $\sigma = \langle s_0, s_1, s_2, \dots \rangle$. We write

$$\sigma[\![a]\!] \text{ for } s_0[\![a]\!]s_1.$$

Thus, an action is evaluated by the first two states of a behavior. ▶

When describing systems, it may be necessary to consider situations where the system performs a step, but no actual changes occur in the state variables. Therefore, we introduce the concept of *stuttering steps.*

Definition 1.16 (Stuttering step)**.** Let a be an action and f be a state function, which is the identity function on all state variables. Hence, $f' = f$ indicates that no change has occurred in state. To accommodate this, we extend the action a to allow for a stuttering step by

$$a \vee (f' = f).$$

This yields

$$\begin{aligned} a \vee (f' = f) &\equiv a \vee (f'(v_1, v_2, \dots, v_n) = f(v_1, v_2, \dots, v_n)) \\ &\equiv a \vee ((v_1', v_2', \dots, v_n') = (v_1, v_2, \dots, v_n)) \\ &\equiv a \vee ((v_1' = v_1) \wedge (v_2' = v_2) \wedge \cdots \wedge (v_n' = v_n)) \end{aligned}$$

and we introduce the notation

$$[a]_{\langle v_1, v_2, \dots, v_n \rangle}$$

for it. This notation allows for an a-step where none of the state variables change. Note that in actual TLA specifications, we would define a tuple of all states variables, i.e., $\mathit{vars} \coloneqq \langle v_1, v_2, \dots, v_n \rangle$, and then use this tuple to describe a stuttering step as $[a]_{\mathit{vars}}$. ▶

In contrast to stuttering steps, we also define actions that strictly require a change in the state for an action a and a state function f. Hence, we want an action given by

$$a \wedge (f' \neq f) \equiv a \wedge (f'(v_1, v_2, \dots, v_n) \neq f(v_1, v_2, \dots, v_n))$$

which we write as

$$\langle a \rangle_f.$$

For convenience, we also define the action *unchanged*, to express situations where there is no change in the value of a state function f.

Definition 1.17 (Unchanged action)**.** Let f be a state function. We define

$$unchanged(f) := [true]_f.$$

Hence, an *unchanged*(f)-step is one in which the value of the state function f does not change. ▶

The Boolean connectives are from propositional logic. Hence, for two formulas ϕ_1 and ϕ_2, it suffices to state

$$\begin{aligned}\sigma[\![\phi_1 \wedge \phi_2]\!] &\equiv \sigma[\![\phi_1]\!] \wedge \sigma[\![\phi_2]\!]\\ \sigma[\![\neg\phi_1]\!] &\equiv \neg\sigma[\![\phi_1]\!].\end{aligned}$$

All other connectives can be derived by applying standard equivalences.
Now to describe the temporal aspect of formulas, we need to define the semantics of $\Box\phi$.

Definition 1.18 (Always *true*)**.** Let ϕ be a formula and $\sigma \in \mathrm{St}^\infty$ be a behavior given by $\sigma = \langle s_0, s_1, s_2, \dots\rangle$. We denote

$$\sigma[\![\Box\phi]\!] \equiv \forall n \in \mathbb{N}.\ \sigma^n[\![\phi]\!].$$

This means that for $\Box\phi$ to be *true* in a behavior σ, it must hold for every suffix σ^n of this behavior. We write $\sigma \vDash \Box\phi$ iff $\sigma[\![\Box\phi]\!] \equiv true$. ▶

By combining the definitions of the $\Box$-operator and predicates on behaviors, we obtain for a predicate P and a behavior σ

$$\begin{aligned}\sigma[\![\Box P]\!] &\equiv \forall n \in \mathbb{N}.\ \sigma^n[\![P]\!]\\ &\equiv \forall n \in \mathbb{N}.\ s_n[\![P]\!].\end{aligned}$$

Hence, σ satisfies $\Box P$, written as $\sigma \vDash \Box P$, iff every state in the behavior σ satisfies P. Similarly, for actions on behaviors, it follows that for an action a and a behavior σ:

$$\begin{aligned}\sigma[\![\Box a]\!] &\equiv \forall n \in \mathbb{N}.\ \sigma^n[\![a]\!]\\ &\equiv \forall n \in \mathbb{N}.\ s_n[\![a]\!]s_{n+1}.\end{aligned}$$

Thus, σ satisfies $\Box a$, written as $\sigma \vDash \Box a$, iff every step of a behavior σ is an a-step. Since $[a]_f$ is simply an action with a special notation, we conclude

$$\begin{aligned}\sigma[\![\Box[a]_f]\!] &\equiv \forall n \in \mathbb{N}.\ \sigma^n[\![[a]_f]\!]\\ &\equiv \forall n \in \mathbb{N}.\ s_n[\![[a]_f]\!]s_{n+1},\end{aligned}$$

for an action a, a behavior σ and a state function f, which is the identity function on all state variables. Therefore, σ satisfies $\Box[a]_f$ iff every step of a behavior σ is an a-step or a stuttering step. We write $\sigma \vDash \Box[a]_f$ iff $\sigma[\![\Box[a]_f]\!] \equiv$ *true*.
For convenience, we describe some more temporal formulas.

Definition 1.19 (Eventually *true*)**.** Let ϕ be a formula and $\sigma \in \mathrm{St}^\infty$ be a behavior given by $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$. We define

$$\Diamond\phi := \neg\Box\neg\phi.$$

Informally, this denotes that ϕ will eventually be *true* during a behavior σ, meaning

$$\sigma[\![\Diamond\phi]\!] \equiv \exists n \in \mathbb{N}.\ \sigma^n[\![\phi]\!].$$ ▶

Definition 1.20 (Infinitely often *true*)**.** Let ϕ be a formula and $\sigma \in \mathrm{St}^\infty$ be a behavior given by $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$. We say

$$\sigma[\![\Box\Diamond\phi]\!] \equiv \forall n \in \mathbb{N}.\ \exists m \in \mathbb{N}.\ \sigma^{n+m}[\![\phi]\!].$$

Hence, an arbitrary behavior σ satisfies $\Box\Diamond\phi$ iff ϕ is *true* infinitely many times along σ. ▶

Definition 1.21 (Eventually always *true*)**.** Let ϕ be a formula and $\sigma \in \mathrm{St}^\infty$ be a behavior given by $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$. We describe

$$\sigma[\![\Diamond\Box\phi]\!] \equiv \exists n \in \mathbb{N}.\ \forall m \in \mathbb{N}.\ m \geq n \Rightarrow \sigma^m[\![\phi]\!].$$

This means that a behavior σ satisfies $\Diamond\Box\phi$ iff there is some point in σ from which onwards ϕ is always *true* in all subsequent states. ▶

Definition 1.22 (Leads to)**.** Let ϕ_1 and ϕ_2 be formulas. We define

$$\phi_1 \leadsto \phi_2 := \Box(\phi_1 \Rightarrow \Diamond\phi_2).$$

Thus, whenever ϕ is *true*, then ψ has to be *true* at the same time or later. ▶

Definition 1.23 (Validity of a formula)**.** Let ϕ be a formula, and St^∞ the collection of all behaviors. We define ϕ to be *valid*, denoted by $\models \phi$, as

$$\models \phi := \forall\sigma \in \mathrm{St}^\infty.\ \sigma[\![\phi]\!].$$ ▶

Lemma 1.1. The $\leadsto$-operator is *transitive*, i.e., for arbitrary formulas ϕ_1, ϕ_2 and ϕ_3

$$\phi_1 \leadsto \phi_2 \wedge \phi_2 \leadsto \phi_3 \Rightarrow \phi_1 \leadsto \phi_3$$

holds.

Proof. Let ϕ_1, ϕ_2 and ϕ_3 be arbitrary formulas, and let σ be an arbitrary behavior. We assume $\phi_1 \leadsto \phi_2 \wedge \phi_2 \leadsto \phi_3$ and aim to show that $\phi_1 \leadsto \phi_3$. Suppose ϕ_1 is *true* at state s_i of σ. By $\phi_1 \leadsto \phi_2$, there must exist a state s_j of σ with $i \leq j$ for which ϕ_2 is *true* onwards. Given $\phi_2 \leadsto \phi_3$ and $\phi_2 \equiv$ *true* from s_j onwards, there exists a state s_k of σ with $j \leq k$ such that ϕ_3 is *true* onwards. Since $i \leq j$ and $j \leq k$ it follows that $i \leq k$. Thus, starting from a state s_i where ϕ_1 holds, there exists a state s_k where ϕ_3 holds. This shows the transitivity of the $\leadsto$-operator. □

With this proof, we have shown the transitivity of the $\leadsto$-operator to be *valid*. Applying this definition to our earlier proof concerning the transitivity of the $\leadsto$-operator, we can express that:

$$\models \phi_1 \leadsto \phi_2 \land \phi_2 \leadsto \phi_3 \Rightarrow \phi_1 \leadsto \phi_3.$$

Before concluding our introduction to the semantics of TLA, we will define the notion of *possible* behaviors.

Definition 1.24 (Possible behaviors)**.** Let ϕ be a formula specifying a system. We denote the set of possible infinite behaviors with respect to ϕ as

$$\mathrm{St}^\infty_\phi := \{\sigma \in \mathrm{St}^\infty \mid \sigma \models \phi\},$$

and the set of possible finite behaviors with respect to ϕ as

$$\mathrm{St}^*_\phi := \{\pi \in \mathrm{St}^* \mid \exists \sigma \in \mathrm{St}^\infty_\phi.\ \pi\sigma \models \phi\}.$$

Hence, St^∞_ϕ and St^*_ϕ are subsets of St^∞ and St^*, respectively, including only (partial) behaviors that satisfy the specification ϕ or can be extended such that they satisfy ϕ. ▶

This concludes our discussion on the semantics of TLA. We now apply these concepts to specify a simple hour clock, which counts from 1 to 12 in an infinite cycle. Since there are obviously steps needed between the transitions of the hour values (e.g., minute or second changes), and our system does not explicitly model these intermediate steps, we allow for stuttering steps to account for these unmodelled transitions.

Example 1.4. We consider the following sets:

$$V_S := \{hr\} \qquad V'_S := \{hr'\} \qquad V_R := \emptyset \qquad L_F := \{+, 1, 12\}$$
$$L_P := \{\mathit{Init}, \mathit{IncrementHour}, \mathit{ResetHour}, \mathit{Next}\}$$

First, we need to describe the initial states. Hence, we define the *Init*-predicate as follows

$$\mathit{Init} := hr \in \{1, 2, \dots, 11, 12\}$$

Next, we need two actions: one to increment the hour variable, and one to reset the variable again once it reaches the maximum value, i.e., 12. Therefore, we define

$$\mathit{IncrementHour} := hr < 12 \land hr' = hr + 1,$$

and

$$\mathit{ResetHour} := hr = 12 \land hr' = 1.$$

The former checks if the *hr* variable is less than 12 and then transitions to a new state where the *hr* variable is incremented by one. The latter resets the

hr variable once the maximum value is reached, transitioning to a new state where $hr = 1$.
Furthermore, we combine both these actions with a disjunction to obtain our *Next*-action, also referred to as *Next*-state relation in the literature on TLA.

$$\mathit{Next} \coloneqq \mathit{IncrementHour} \lor \mathit{ResetHour}$$

To finalize our system specification, given by a formula ϕ, we combine the *Init*-predicate and the *Next*-action (including stuttering steps) into the standard form:

$$\phi \coloneqq \mathit{Init} \land \Box[\mathit{Next}]_{\langle hr \rangle}.$$

According to the semantics we defined, for a behavior $\sigma \in \mathrm{St}^\infty$, given by $\sigma = \langle s_0, s_1, s_2, \dots \rangle$, we conclude

$$\begin{aligned} \sigma \in \mathrm{St}^\infty_\phi &\iff \sigma \vDash \phi \\ &\iff \sigma \vDash \mathit{Init} \land \sigma \vDash \Box[\mathit{Next}]_{\langle hr \rangle} \\ &\iff \sigma[\![\mathit{Init}]\!] \land \sigma[\![\Box[\mathit{Next}]_{\langle hr \rangle}]\!] \\ &\iff s_0[\![\mathit{Init}]\!] \land \forall n \in \mathbb{N}.\ s_n[\![[\mathit{Next}]_{\langle hr \rangle}]\!]s_{n+1}. \end{aligned}$$

Hence, the set St^∞_ϕ contains only behaviors where the first state satisfies the *Init*-predicate, and every step along the behavior is either a *Next*-step or a stuttering step.

Having illustrated how we can specify the behavior of a system using TLA, we will now continue by examining the properties that are relevant to us.

1.1.3 System Properties

We categorize the properties that we check on systems into two types: safety and liveness. We now provide formal definitions for both, following [11], while incorporating the notions of TLA.

Safety

A *safety* property asserts that something must *never* happen [3].

Definition 1.25 (Safety property)**.** Let ϕ be a formula that specifies a system. We say that a formula ψ is a safety property for the system iff

$$\forall \sigma \in \mathrm{St}^\infty_\phi.\ (\sigma \nvDash \psi \Rightarrow (\exists i \in \mathbb{N}.\ (\forall \pi \in \mathrm{St}^\infty_\phi.\ \sigma_i \pi \nvDash \psi))).$$

Hence, if a behavior σ does not satisfy ψ, there must exist a prefix σ_i such that all possible continuations from σ_i will also fail to satisfy ψ. In other words, a safety property is violated as soon as there is a single identifiable state in the behavior where ψ does not hold, and this violation persists in all subsequent states. To further clarify this concept, we consider an alternative description by removing the negations in the definition above. This yields

$$\forall \sigma \in \mathrm{St}_\phi^\infty .\ ((\forall i \in \mathbb{N}.\ (\exists \pi \in \mathrm{St}_\phi^\infty .\ \sigma_i \pi \vDash \psi)) \Rightarrow \sigma \vDash \psi).$$

This restatement suggests that if, for every prefix σ_i, there exists a continuation π such that the combination $\sigma_i \pi$ satisfies ψ, then the entire behavior σ satisfies ψ. Hence, if σ satisfies ψ, then there cannot exist a single state in σ that violates ψ. ▶

An example for such a safety property would be mutual exclusion, which we formally define in Subsection 2.1.2.

Liveness

A *liveness* property asserts that something *eventually* happens [3].

Definition 1.26 (Liveness property)**.** Let ϕ be a formula that specifies a system. We call a formula ψ a liveness property for the system iff

$$\forall \pi \in \mathrm{St}_\phi^* .\ (\exists \sigma \in \mathrm{St}_\phi^\infty .\ \pi\sigma \vDash \psi).$$

In this definition, we consider a partial behavior π to be *live* with respect to ψ iff there is a behavior σ from π onwards such that $\pi\sigma$ satisfies ψ. Hence, a liveness property is one for which every partial behavior that is possible for ϕ is live. ▶

A classic example of a liveness property is starvation freedom, which in general states that a process makes progress infinitely often.

Since TLA allows for stuttering steps, any finite behavior π can be extended to an infinite behavior σ by appending an infinite sequence of stuttering steps. This may lead to behaviors where the system appears to make no progress, i.e., there are infinitely many steps in which none of the state variables change, which may not align with what we intend to model. While we cannot guarantee that such behaviors will not occur in real systems, we sometimes want to omit behaviors where no progress is observed to meaningfully reason about liveness properties. This omission can be managed by *fairness* constraints [4].

Fairness

In general, we distinguish between *weak* and *strong* fairness. To verify the algorithms of Chapter 2, we will make use of both. According to [4], weak and strong fairness can be formally defined as:

Definition 1.27 (Weak and strong fairness constraints)**.** Let a be an action. We define a *weak fairness* constraint on a as

$$\mathit{WF}_f(a) := \Box(\Box \mathit{enabled}(\langle a \rangle_f) \Rightarrow \Diamond \langle a \rangle_f)$$

where f is a state function symbol. Informally, weak fairness asserts that an action a, which ultimately alters a state function f, must eventually occur if it remains *enabled* from a certain point onwards. Hence, it cannot be ignored indefinitely if it remains enabled.
Furthermore, we define a *strong fairness* constraint on a as

$$SF_f(a) \coloneqq \Box(\Box\Diamond enabled(\langle a\rangle_f) \Rightarrow \Diamond\langle a\rangle_f)$$

In contrast to weak fairness, strong fairness asserts that an action a, which again ultimately alters a state function f, must occur infinitely often if it is *enabled* infinitely often. Hence, if a is enabled repeatedly, it must eventually occur, even if there are periods when it is not continuously enabled. ▶

From the definitions, it is clear that weak fairness is implied by strong fairness. These fairness restrictions are not limited to actions structured like $\langle a\rangle_f$. However, we adhere to the original definitions found in the literature on TLA. By incorporating fairness constraints, we restrict the possible behaviors to *fair* behaviors with respect to the given constraints [1].

1.2 TLA$^+$

Having established the rigorous foundation of TLA, we now study the specification language TLA$^+$ which builds upon TLA. TLA$^+$ implements the generic definition of TLA, given in Section 1.1, by using first-order logic and the Zermelo-Fränkel set theory with choice (ZFC). Since TLA$^+$ is based on TLA, we have already covered most of the syntax and semantics in Section 1.1. Therefore, we will only introduce language concepts following [4, 12] that are necessary to comprehend the specifications given in Chapter 2. For a complete and thorough introduction to TLA$^+$, we refer the reader to [12].

1.2.1 Modules

In general, TLA$^+$ specifications are structured as modules. We distinguish between TLA$^+$ standard modules and user-defined modules. An example of the former is the `Naturals` module, which inherits the definition of the *Nat* set, representing the natural numbers, from the `Peano` module. Through this concept of modularization, it is possible to specify complex systems in a way that remains easily understandable. We will now examine a simple example of an hour clock, given in Figure 1.1, to discuss to most important parts of a module.
The name of a module, e.g. *HourClock*, is always stated in its header. To import a standard module into a user-defined module, we utilize the `EXTENDS` keyword. By using the `VARIABLE` keyword, we define a state variable, meaning $hr \in V_S$. The usage of the `CONSTANT` keyword defines a constant whose value can be set in the model, for example, by using a TLC configuration file.

MODULE *HourClock*

EXTENDS *Naturals*

VARIABLE *hr*

CONSTANT *twelveHrs*

$Init \triangleq hr \in 1 \mathrel{..} twelveHrs$

$IncrementHour \triangleq hr < twelveHrs \land hr' = hr + 1$

$ResetHour \triangleq hr = twelveHrs \land hr' = 1$

$Next \triangleq IncrementHour \lor ResetHour$

$Spec \triangleq Init \land \Box[Next]_{hr}$

$TypeOK \triangleq hr \in 1 \mathrel{..} twelveHrs \land hr \in Nat$

Figure 1.1: Example of a module

Next, we encounter the *Init*-predicate, which expresses conditions on the initial states. Following that, we have the *IncrementHour* and *ResetHour*-actions, which, connected by a disjunction, form the *Next*-action. To conclude the TLA$^+$ specification, we have the usual form of a TLA$^+$ specification. We note that such modules usually are expressed using only ASCII characters. Here, we made use of the TLATEX typesetter to obtain a pretty-printed version.

1.2.2 Additional Notions

As already mentioned, TLA$^+$ extends TLA with additional notions and data structures that simplify describing concurrent systems. We now introduce a small fraction of these additions following [4, 12].

Sets

As already mentioned, TLA$^+$ is based on Zermelo-Fränkel set theory, in which every value is a set. Sets can be defined either explicitly or implicitly in TLA$^+$ specifications. To explicitly define a set, one can either enumerate all elements, e.g.

$$S \coloneqq \{1, 2, 3, 5, 7\}$$

or give a start and end value with two dots in between, which describes a set consisting of a consecutive sequence of numbers, e.g.

$$S \coloneqq m \mathrel{..} n = \{m, m+1, m+2, \ldots, n-2, n-1, n\}$$

for $m, n \in N$ and $m \leq n$. Now to implicitly define sets, we can write

$$S \coloneqq \{x \in T : p\}$$

which expresses that S includes all $x \in T$ that satisfy p. Hence, it follows that $S \subseteq T$. Another way for an implicit definition is by writing

$$S \coloneqq \{e : x \in T\}$$

where $x \in T$ is a bound variable in the expression e, which characterizes the elements of S. For example, one could use $\{2n : n \in \mathbb{N}\}$ to specify the set of all even natural numbers.
To make use of these sets inside of specifications, TLA$^+$ offers the following built-in set operators

$$\in \quad \notin \quad \cup \quad \cap \quad \subset \quad \setminus \quad \texttt{UNION} \quad \texttt{SUBSET}$$

where all of these are defined as usual, except for the unary operators `UNION` and `SUBSET`.

Definition 1.28 (`UNION`-operator)**.** Let S be a set. The `UNION`-operator is given by

$$\texttt{UNION}\ S \coloneqq \{x \in T \mid \exists T \in S\}$$

Intuitively, this means that `UNION` S contains all elements of the sets included in S. ▶

Definition 1.29 (`SUBSET`-operator)**.** Let S be a set. We define the `SUBSET`-operator as

$$\texttt{SUBSET}\ S \coloneqq \mathcal{P}(S) = 2^S = \{X \mid X \subseteq S\}$$

Hence, the `SUBSET`-operator is the powerset of S. ▶

Furthermore, the `FiniteSet` standard module implements the *Cardinality(S)* and *isFiniteSet(S)*-operators, for an arbitrary set S, with their expected semantics.

Functions

In TLA$^+$, functions are assumed to be primitive and are used as the foundation for multiple data structures. We denote the set of functions with a domain S and a codomain T as $[S \mapsto T]$. Given an expression e, the application of a function f to this expression is written as $f[e]$. Functions are defined explicitly in TLA$^+$ by

$$f \coloneqq [x \in \mathit{dom}\, f \mapsto f[x]]$$

Additionally, it is possible to override certain mappings using the `EXCEPT` clause which we will illustrate by an example.

Example 1.5. Let f be a function defined as $f \coloneqq [n \in \mathbb{N} \mapsto n+1]$. We define a new function g to override the mapping of a specific $k \in \mathbb{N}$ with a new image $g[k] = a$, where $a \in \mathbb{N}$ by

$$g \coloneqq [f\ \texttt{EXCEPT}\ ![k] = a]$$

Hence, g is characterized as

$$g[n] := \begin{cases} a, & \text{if } n = k \\ f[n], & \text{otherwise} \end{cases}$$

As usual, we say that two functions f, g are equal, written as $f = g$, iff

$$\mathit{dom}\ f = \mathit{dom}\ g \land \forall x \in \mathit{dom}\ f.\ f[x] = g[x]$$

Tuples

An n-tuple $\langle e_1, \dots, e_n \rangle$ is a function with the set $\{1, \dots, n\}$ as its domain, defined as

$$\langle e_1, \dots, n \rangle := [i \in \{1, \dots, n\} \mapsto e_i]$$

Hence, to access the i-th entry of an n-tuple, we have $\langle e_1, \dots, n \rangle[i] = e_i$ for $1 \leq i \leq n$. Since tuples are defined as functions, two tuples with the same elements but different nesting are not equal, e.g.

$$\langle a, b, c \rangle \neq \langle a, \langle b, c \rangle \rangle$$

Now to change a single value e_i of a tuple, we create a new tuple with an alternative mapping on the i-th position, as we explained while discussing functions.

Example 1.6. Let x be a tuple given by $x := \langle a, b, c \rangle$. To replace the second element b with another element d, we write

$$x := [x\ \texttt{EXCEPT}\ ![2] = d]$$

where 2 refers to the index of the element to be replaced. This results in a new tuple $\langle a, d, c \rangle$.

Strings

A string of length n is an n-tuple of characters. For example, the string "hello world!" is represented as

$$\langle \text{“h”}, \text{“e”}, \text{“l”}, \text{“l”}, \text{“o”} \text{“\_”}, \text{“w”}, \text{“o”}, \text{“r”}, \text{“l”}, \text{“d”} \text{“!”} \rangle$$

Note that the individual characters do not have a fixed meaning; they are only defined to be different to each other. This means that it is possible to assign different encodings to them, such as `ASCII`, for example.

1.2.3 Toolbox

Throughout the years, multiple tools have been developed to make full use of the capabilities of TLA$^+$. The most recent example is a symbolic model checker called “Apalanche”, which was developed by Konnov, Kukovec, and Tran at Inria and TU Vienna [13]. In this section, we will give short descriptions about a few of these tools that are part of the official TLA$^+$ toolbox and which we used during the creation of this thesis. For a more detailed explanation of each tool, we refer the reader to [12] and [14]. Note that we do not mention the TLC model checker here, as it will be thoroughly discussed in Subsection 1.3.2.

Syntactic Analyzer

The Syntactic Analyzer (SANY) is a Java program to parse and validate the structure of TLA$^+$ specifications. It was written by Jean-Charles Grégoire and David Jefferson and works either as a standalone tool or as a front end in the TLC model checker. SANY distinguishes between syntactic and semantic errors and provides hints to the user about the location of these errors. To check a given specification for errors, one simply calls:

```
java -jar <path-to-toolbox>/tla2tools.jar
     <tla-specification-file>
```

Listing 1.1: SANY example usage

Note that it might be necessary to change the `MANIFEST.MF` located inside the `tla2tools.jar` to address the corresponding Java `class`.

TLATEX Typesetter

TLATEXis a Java program used to properly typeset TLA$^+$ specifications given in simple ASCII formatting. It is a part of the TLA$^+$ toolbox.
To use the typesetter for a given LATEX file, it needs to include a `\tla`-environment filled with a TLA$^+$ module. The `\tla`-environment will be replaced by a typeset version after executing:

```
java -jar <path-to-toolbox>/tla2tools.jar
     -out <latex-out-file>
     <latex-input-file>
```

Listing 1.2: TLATEX Typesetter example usage

Alternatively, one can use the `Produce PDF Version` command in the user interface of the TLA$^+$ toolbox.

1.3 Model Checking

Before discussing TLC, we give a brief introduction to model checking, following [1]. Model checking is a verification technique used to verify the correctness of finite state concurrent systems by checking if they satisfy a set of defined properties. Since the systems are finite, model checking can be executed automatically, which is one of its main advantages over deductive verification. During this verification process, the whole state space of the system will be checked, resulting in a *yes* or *no* answer, depending on whether the system satisfies the specified properties. This process is illustrated in Figure 1.2. For many systems, this finite state space is sufficient; for others, it is possible to bound the state space to a finite one in order to find errors.
In general, the process of model checking consists of the following steps:

- **Modeling**: First, the system has to be converted into a formalism, the so-called *model*, supported by the model checking tool of choice. If the system is too complex, it might be necessary to use an abstraction of it.
- **Specification**: This task involves stating the properties that the system must satisfy. These properties are usually expressed using a temporal logic like LTL or CTL. It is important to ensure *completeness* of the specification, meaning that all desired properties are specified, as this cannot be checked during model checking.
- **Verification**: While most of the verification is performed automatically, there is still a need for human interaction, for example, when handling potential errors. When a model checker encounters a violation of the given specification, it provides a counterexample, also called an error trace, to fully comprehend the origin of the error. These counterexamples are invaluable for finding flaws in complex system designs.

Note that the term "specification" has different meanings in the context of TLA and model checking. A TLA specification represents the model, whereas the properties, expressed using TLA, that a model must satisfy are understood as the specification in terms of model checking.

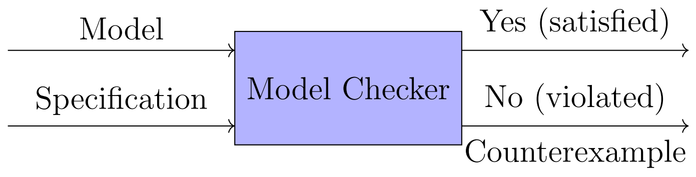

Figure 1.2: Simplified model checking process

1.3.1 State Space Explosion

While model checking is arguably one of the most promising verification techniques, it suffers from the *state space explosion* problem, which we will briefly introduce following [2].
Consider a software system with a set of variables $x \in V_S$. Each of this variables has a domain $dom(x)$, which represents all possible values that it can take. To memorize the current location of an execution, we need a *pc* (program counter) variable with values in the set of possible locations L. This leads to a possible state space given by

$$|L| \cdot \prod_{x \,\in\, V_S} |dom(x)|.$$

Hence, the number of states grows *exponentially* with the number of state variables. With n variables, where each has a domain of k possible values, the number of states has an upper limit of k^n. Since software systems may have dozens of variables, the state space might become too large for model checking.
If we now consider concurrent executions given by

$$P = P_1 \mid\mid \ \ldots \ \mid\mid P_n$$

where P_i is a single concurrent process, the problem becomes even more prominent. Let S_i denote the state space of a concurrent process P_i. Since each possible state of P is now the Cartesian product of its local states, i.e. $S = S_1 \times \ \ldots \ \times S_n$, we get

$$|S| = |S_1| \cdot \ \ldots \ \cdot |S_n|$$

as the number of total states of P. Hence, even simple programs executed concurrently lead to a exponentially large number of states.

There are multiple approaches to combat this problem, such as symbolic representation, partial order reduction and many more. For more details about these techniques, we refer the reader to [1] and [2].

1.3.2 TLC

TLC is a model checker for TLA$^+$ specifications, developed by Yuan Yu in 1999. Since TLA$^+$ was not initially designed to be used for model checking, TLC is only capable of verifying a subset of the specifications that can be described using TLA$^+$. This subset consists of specifications that have the standard form

$$Init \wedge \Box[Next]_{vars} \wedge Temporal,$$

where *Init* is the initial predicate, *Next* is the next-state action, *vars* is the tuple of all state variables and *Temporal* is a temporal formula that usually

specifies a liveness condition by the use of fairness constraints, as discussed in Section 1.1.3. Note that the *Temporal* formula can be omitted if it is not needed. There are some additional restrictions on the specifications that TLC can handle. However, Yu and Lamport argue that TLC can manage most of the specifications of actual system designs [15, 12]. For further details about these restrictions, we refer the reader to [15] and [12].
Unlike other popular model checkers like SPIN or Uppaal, TLC stores the verification progress on disk and uses the main memory only as a cache. This approach mitigates the space limitation during model checking, though it implies reduced performance. As stated in [15], performance was not one of the main concerns during the development of TLC.
TLC takes a TLA$^+$ module and a configuration file as input, where the module contains the system specification and the configuration file all necessary information for TLC to check the given system. This configuration file may contain multiple declarations:

- `SPECIFICATION`: Name of the module specifying the system
- `INIT`: Name of the *Init* predicate
- `NEXT`: Name of the next-state action
- `INVARIANT`: Name of an invariant to check, which must be a state predicate
- `PROPERTY`: Name of a property to check
- `CONSTANT`: Assignment of values to the constants of the specification
- `CONSTRAINT`: Name of a predicate to restrict the set of reachable states
- `ASSUME`: Name of a predicate that describes assumptions within the specification

In general, TLC supports two different execution modes: the *model checking mode* and the *simulation mode*. In short, the simulation mode randomly generates behaviors with a fixed length and checks these against the given properties. Hence, this approach does not exhaustively check all reachable states, in contrast to the model checking mode. For the model checking mode, the system needs to be finite, which can either be handled by constants or by making use of the `CONSTRAINTS` keyword [12]. We will give an overview on how the model checking mode of TLC works for safety properties, based on [15]. For a more thorough explanation and details on checking liveness properties, we refer the reader to [12] and [16].

During model checking a safety property ϕ, TLC manages two data structures:

- A set *seen* of all states that are known to be reachable, and
- a FIFO queue *sq* which contains elements of *seen* whose successor states have not been checked yet.

Initially, TLC generates all states that satisfy the *Init*-predicate and places them into *seen* and *sq*. Following this, the *Next*-action is rewritten into a disjunction of as many simple subactions as possible. We denote the set of all such subactions as A. Additionally, we define the conjunction of all predicates given by the `CONSTRAINT` keyword as C. Now, TLC starts multiple worker threads, each repeatedly executing pseudocode given in Algorithm 1.

Algorithm 1 TLC Model Checking Algorithm for Safety Properties [15]

```
    Assumptions: $sq \neq \emptyset$ initially, all elements of $A$ are valid actions
 1: while $sq \neq \emptyset$ do
 2:     $s := sq$.dequeue()
 3:     $anySuccessors :=$ false
 4:     for all Subaction $a \in A$ do
 5:         $T_a :=$ generateSuccessorStates($s$, $a$)
 6:         if $T_a \neq \emptyset$ then
 7:             $anySuccessors :=$ true
 8:             for all $t \in T_a$ do
 9:                 if $t \notin seen$ then
10:                     if $t \vDash \phi$ then
11:                         $seen := seen \cup \{t\}$
12:                         if $t \vDash C$ then
13:                             $sq$.enqueue($t$)
14:                         end if
15:                     else
16:                         return "violation"
17:                     end if
18:                 end if
19:             end for
20:         end if
21:     end for
22:     if $\neg anySuccessors$ then
23:         return "deadlock"
24:     end if
25: end while
```

We note that on line 5, a possible successor state t is one for which (s, t) is an a-step. In order to construct a counterexample, TLC attaches a reference pointing from t to s when adding t to *seen* on line 11. Clearly, this algorithm only terminates if the state space of the system is finite.

2. Verifying Mutual Exclusion

In this chapter, we will analyze and verify two variants of an algorithm introduced by Szymanski [17] to solve the mutual exclusion problem formalized by Dijkstra [18] and Lamport [19]. We begin by expressing the models and their desired properties in TLA$^+$. Following this, we will utilize the TLC model checker to verify whether the algorithms satisfy these properties. Finally, for each algorithm, we will discuss our findings and compare them to the results of Spronck and Luttik, who modeled these two variants using mCRL2 [20, 21].

2.1 Methodology

This section describes the steps we followed to verify the two variants of the mutual exclusion algorithm. It aims at being precise enough to give the reader the ability to reproduce the verification steps and apply the given methodology to other similar concurrent algorithms without additional literature.

2.1.1 Model

To verify the algorithms given in Section 2.2 and Section 2.3, we first need to create models based on their pseudocode. We do this by applying the methods stated in [1] while making minor adaptations to obtain a valid TLA$^+$ specification that follows best practices. For our use case, we assume the model that we want to express is

$$P_1 \mid\mid P_2 \mid\mid \dots \mid\mid P_n$$

where $P_1, P_2, \dots, P_n$ are processes, each executing the same algorithm. Hence, we model a concurrent program consisting of n processes with asynchronous interleaving. We start by labeling the pseudocode of such an algorithm, following the labeling transformation of [1]. Through this transformation, we obtain the set of possible *program locations*.

Definition 2.1 (Labeling transformation)**.** Let P be a statement of an algorithm. We define the *labeled* statement $P^{\mathcal{L}}$ as:

- If P is an *atomic statement*, then

$$P^{\mathcal{L}} := P$$

- If $P = P_1;\ P_2$ (*sequential composition*), then

$$P^{\mathcal{L}} := P_1^{\mathcal{L}};\ l'' : P_2^{\mathcal{L}}$$

- If $P = \text{if } b \text{ then } P_1 \text{ else } P_2$ (*conditional*), then

$$P^{\mathcal{L}} := \text{if } b \text{ then } l_1 : P_1^{\mathcal{L}} \text{ else } l_2 : P_2^{\mathcal{L}} \text{ endif}$$

- If $P =$ while b do P_1 endwhile (*while loop*), then

$$P^{\mathcal{L}} \coloneqq \text{ while } b \text{ do } l_1 : P_1^{\mathcal{L}} \text{ endwhile}$$

where l_1, l_2, l'' are new unique labels. Additionally, we mark the beginning and ending of the provided program by m and m' respectively. ▶

We note that we neglected the case where P is a *parallel composition*, since specifying the concurrent nature of systems is handled slightly different in TLA$^+$. We will describe this in more detail later on.

After labeling an algorithm, we want to obtain the set of transitions. In terms of TLA$^+$, this set of transitions describes all possible behaviors of our algorithm, starting from the initial states. Therefore, we first describe the set of initial states by

$$\mathit{Init} \coloneqq \bigwedge_{i=1}^{n} \mathit{pre}_i(V_S) \wedge \mathit{pc}_i = m$$

where $\mathit{pre}_i(V_S)$ is a predicate representing conditions on the initial values of the local variables of the i-th process, and pc_i equals the entry label m. Hence, the set of initial states S_0 is defined as

$$S_0 \coloneqq \{s \in \text{St} \mid s \vDash \mathit{Init}\}.$$

Afterwards, we apply the translation procedure $\mathcal{C}(l, P, l')$ of [1] inductively on the structure of P to obtain the possible transitions within our algorithm. This translation procedure is defined as:

Definition 2.2 (Translation procedure)**.** Let P be a labeled statement of an algorithm, l be its entry label and l' be its exit label. We define $\mathcal{C}(l, P, l')$ as:

- If $P = (v_i \coloneqq e)$, then

$$\mathcal{C}_i(l, P, l') \coloneqq \mathit{pc}_i = l \wedge \mathit{pc}_i' = l' \wedge v_i' = e \wedge \mathit{unchanged}(V_S \setminus \{v, \mathit{pc}\})$$

- If $P = \mathit{skip}$, then

$$\mathcal{C}_i(l, P, l') \coloneqq \mathit{pc}_i = l \wedge \mathit{pc}_i' = l' \wedge \mathit{unchanged}(V_S \setminus \{\mathit{pc}\})$$

- If $P = P_1; l''$: P_2, then

$$\mathcal{C}_i(l, P, l') \coloneqq \mathcal{C}_i(l, P_1, l'') \vee \mathcal{C}_i(l'', P_2, l')$$

- If $P =$ if b then l_1: P_1 else l_2: P_2 endif, then

$$\begin{aligned}\mathcal{C}_i(l, P, l') \coloneqq\ & (\mathit{pc}_i = l \wedge \mathit{pc}_i' = l_1 \wedge b \wedge \mathit{unchanged}(V_S \setminus \{\mathit{pc}\}) \\ & \vee (\mathit{pc}_i = l \wedge \mathit{pc}_i' = l_2 \wedge \neg b \wedge \mathit{unchanged}(V_S \setminus \{\mathit{pc}\}) \\ & \vee \mathcal{C}_i(l_1, P_1, l') \\ & \vee \mathcal{C}_i(l_2, P_2, l')\end{aligned}$$

- If $P =$ while b do $l_1\colon P_1$ endwhile, then

$$\begin{aligned}\mathcal{C}_i(l, P, l') \coloneqq{} & (pc_i = l \land pc_i' = l_1 \land b \land \mathit{unchanged}(V_S \setminus \{pc\}) \\ & \lor (pc_i = l \land pc_i' = l' \land \neg b \land \mathit{unchanged}(V_S \setminus \{pc\}) \\ & \lor \mathcal{C}_i(l_1, P_1, l),\end{aligned}$$

- If $P =$ wait(b), then

$$\begin{aligned}\mathcal{C}_i(l, P, l') \coloneqq{} & (pc_i = l \land pc_i' = l \land \neg b \land \mathit{unchanged}(V_S \setminus \{pc\}) \\ & \lor (pc_i = l \land pc_i' = l' \land \neg b \land \mathit{unchanged}(V_S \setminus \{pc\})\end{aligned}$$

where pc_i is the program counter variable of process i, b is a predicate representing the condition of if and while constructs, V_S is the set of all variables and $\mathit{unchanged}(V_S)$ is the *unchanged*-action as defined in Subsection 1.1.2. ▶

Note that we slightly adjusted the original definition of [1] by using the *unchanged*-action of TLA to indicate that some or all variables of V_S do not change. Additionally, we added the index of a process to the program counter for all cases. Since $pc \in V_S$ in the case of TLA$^+$ and due to the way how these process-specific variables are encoded in the specification later on, we also changed the parameters of the *unchanged*-action. Again, this is due to the way concurrency is usually specified in TLA$^+$.

Now that we obtained the set of all transitions, we can define an action *Algorithm* as

$$\mathit{Algorithm}(i) \coloneqq \mathcal{C}_i(m, P, m')$$

where m and m' mark the entry and exit of P. To incorporate the semantics of multiple concurrent processes following [12], we define our *Next*-action as

$$\mathit{Next} \coloneqq \exists p \in 1 \,..\, n \ :\ \mathit{Algorithm}(p)$$

where n is the number of processes. Therefore, our TLA$^+$ specification representing the behavior we want to model is given by

$$\mathit{Spec} \coloneqq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$$

where *vars* is a tuple containing all program variables without any indices. We can omit these indices since the variables themselves are modeled as tuples, where each index addresses a certain value, as discussed in Section 1.2.2. This is why we did not incorporate the indices while stating the variables that did not change, since the change of a single value inside a tuple is seen as a change to the whole tuple.

2.1.2 Properties

For each algorithm, we check if it satisfies both *mutual exclusion* and *starvation freedom.*

Definition 2.3 (Mutual exclusion)**.** Let P be the set of all processes. We define the safety property *mutual exclusion* as

$$\forall i, j \in P.\ (i \neq j) \Rightarrow \neg(\mathit{insideCrit}(i) \land \mathit{insideCrit}(j)),$$

where $\mathit{insideCrit}(i)$ is a predicate stating that a process i is currently inside its critical section. ▶

Informally, this means that no two processes can be inside their critical sections at the same time.

Definition 2.4 (Starvation freedom)**.** Let P be the set of all processes. We define the liveness property *starvation freedom* as

$$\forall i \in P.\ \Box(\mathit{hasIntention}(i) \Rightarrow \Diamond \mathit{insideCrit}(i)),$$

where $\mathit{hasIntention}(i)$ is a predicate stating that a process i wants to enter its critical section. ▶

In simpler words, this ensures that if a process declares its intention to enter its critical section, it will eventually be able to do so. Even though our main focus lies within verifying mutual exclusion, we check for starvation freedom as well, since it is another important aspect of mutual exclusion algorithms [17]. As mentioned in Section 1.1.3, to check liveness properties in a meaningful way, we have to include fairness constraints in our specification. Since these constraints depend on the concrete specification, we will discuss them further in Section 2.2 and Section 2.3.

2.1.3 Verification

Now that we modelled our algorithm and specified its properties, we prepare the verification process. To utilize the TLC model checker we have two options: we can either rely on the graphical user interface of the toolbox or start the verification using its command line interface. Since we want the verification process to be as transparent as possible, we used the command line interface with self-written configuration files. To create these configuration files, we use the parameters discussed in Subsection 1.3.2. To keep the state space manageable, we limit ourselves to two, three, four and five concurrent processes.

```
CONSTANT n = 2
SPECIFICATION Spec
INVARIANT TypeInvariant
INVARIANT MutualExclusion
PROPERTY StarvationFreedom
```

Listing 2.1: TLC configuration file for $n = 2$

To start the verification process we execute the following command:

```
java -jar <path-to-toolbox>/tla2tools.jar
     -config <config-file>
     -workers <number-of-threads>
     <spec-file>
```

Listing 2.2: TLC example usage

We note that, as discussed in Subsection 1.3.2, TLC automatically checks for deadlocks as well, even though it is not mentioned in the configuration.
After executing the command in Listing 2.2 on a given specification and configuration, the model checker states the results of the verification process and provides additional information, such as the size of the state space. An excerpt of such an output for a successful verification is illustrated in Listing 2.3.

```
...
Model checking completed. No error has been found.
Estimates of the probability that TLC did not check all
   reachable states
because two distinct states had the same fingerprint:
calculated (optimistic):  val = 5.0E-16
193 states generated, 96 distinct states found, 0 states
   left on queue.
The depth of the complete state graph search is 16.
...
```

Listing 2.3: TLC verification output on success

In case TLC finds a violation of one of the given properties, it provides a counterexample:

```
...
Error: Invariant MutualExclusion is violated.
Error: The behavior up to this point is:
State 1: <Initial predicate>
/\ doorIn = <<0, 0, 0>>
/\ doorOut = <<0, 0, 0>>
/\ pc = <<"m", "m", "m">>
/\ intent = <<0, 0, 0>>
...
State 33: <l10 line 140, col 9 to line 143, col 48 of
   module SzymanskiFlagBits>
/\ doorIn = <<0, 1, 1>>
/\ doorOut = <<1, 1, 1>>
/\ pc = <<"l15", "l11", "l11">>
/\ intent = <<0, 1, 1>>
...
```

Listing 2.4: TLC verification counterexample

Experimental Setup

For the results reported in Subsection 2.2.2 and Subsection 2.3.2, we used the TLA$^+$ toolbox in version 1.7.1 on an Intel-based consumer-grade machine running Ubuntu 22.04.3 LTS. This machine features an Intel i7-13700H processor with 20 cores at 3.7-5.0 GHz (24 MB L3 cache) and 32 GB of memory. The exact versions for each component of the TLA$^+$ toolbox are listed in Table 2.1.

Software	Version
SANY Synactic Analyzer	2.2
TLC Model Checker	2.16
TLATEX Typesetter	1.0

Table 2.1: Versions of the TLA$^+$ toolbox

2.2 Szymanski's Flag Algorithm

Throughout this section, we will study Szymanski's original flag algorithm (integer variant) following [17]. In general, we differentiate between the non-critical and the critical section of a concurrent process. As mentioned in Subsection 2.1.2, each algorithm designed to tackle the mutual exclusion problem aims to ensure that no two processes are executing their critical section simultaneously. To accomplish this, algorithmic solutions to this problem surround the critical section with a preceding *prologue* and a succeeding *epilogue* [17]. This concept is illustrated in Figure 2.1.

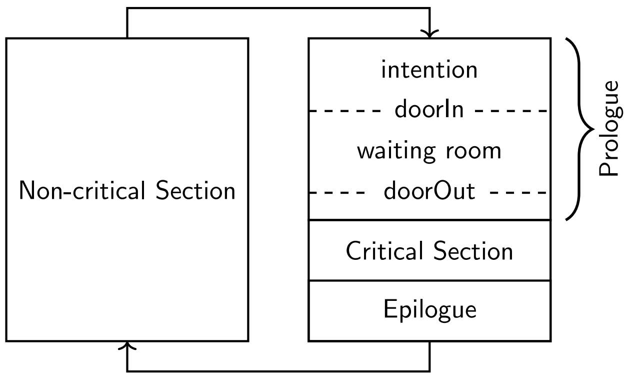

Figure 2.1: Structure and idea of the algorithm [17]

In this concrete algorithm, each process owns a *flag* variable which indicates its current status in relation to the other processes. This *flag* variable is visible to every other process and encodes the semantics stated in Table 2.2.

flag value	Meaning
0	Currently in its non-critical section
1	Showing intention to enter its critical section
2	Waiting for others to get through the *doorIn*
3	Just passed the *doorIn*
4	Just crossed the *doorOut*

Table 2.2: Meaning of the *flag* variable

To convey the general idea of the algorithm given in Algorithm 2, imagine the prologue to be a waiting room with two doors named *doorIn* and *doorOut*. We assume the *flag* variable of each process is initially set to 0. First, each process enters the prologue and sets its *flag* variable to 1, indicating its *intention* to enter the critical section. Since the *doorIn* is open at the beginning, the condition on line 2 is satisfied, and the first process to make progress sets its *flag* variable to 3, indicating it has just passed the *doorIn*. If there are more processes that show their intention to enter their critical section (line 4), each sets its *flag* variable to 2 to give other processes the chance to enter the waiting room. This continues until the last process showing its intention enters the waiting room and sets its *flag* variable to 4 (line 10), as there are no others waiting. This action closes the *doorIn* and opens the *doorOut*. Now, all other processes set their *flag* value to 4 as well, indicating that all processes that were previously in the waiting room have crossed the *doorOut*. Finally, the process with the lowest process identifier enters the critical section, while all other processes have to wait (line 11). This cycle continues until all processes that were in the waiting room before have executed their critical section on line 12. We note that every process that finishes its critical section checks if there is another process in the waiting room that has not yet made its way through the *doorOut* yet (line 13). If there is no such process, it sets its *flag* variable to 0. This action closes the *doorOut* and opens the *doorIn* again, allowing the next batch of processes to progress and restarting the cycle of the algorithm.

Algorithm 2 Szymanski's flag algorithm with integer [20]

```
            Assumption: $\forall i \in \{1, \dots, n\}.\ flag[i] = 0$
($m$)       1:  $flag[i] := 1$
($l_0$)     2:  await $\forall j.flag[j] < 3$
($l_1$)     3:  $flag[i] := 3$
($l_2$)     4:  if $\exists j.flag[j] = 1$ then
($l_4$)     5:      $flag[i] := 2$
($l_6$)     6:      await $\exists j.flag[j] = 4$
            7:  else
($l_5$)     8:      skip
            9:  end if
($l_3$)     10: $flag[i] := 4$
($l_7$)     11: await $\forall j < i.flag[j] < 2$
($l_8$)     12: critical section
($l_9$)     13: await $\forall j > i.flag[j] < 2 \lor flag[j] > 3$
($l_{10}$)  14: $flag[i] := 0$
```

Note that, as covered above, the sequence of flag values during a single execution does not follow a strict numerical order. The possible transitions between these values are illustrated in Figure 2.2.

$$\begin{tikzcd}
0 \arrow[r] & 1 \arrow[rr, bend left] & 2 \arrow[rr, bend right] & 3 \arrow[l] \arrow[r] & 4 \arrow[llll, bend right]
\end{tikzcd}$$

Figure 2.2: Possible transitions of the flag variable

2.2.1 Specification

In this section, we will discuss the most interesting parts of the TLA$^+$ specification obtained by following the methodology stated in Section 2.1. For the complete specification, we refer the reader to Appendix A.1.

To specify the behavior of the integer variant of Szymanski's flag algorithm, we created the user-defined module `SzymanskiFlagInteger`. This module features two state variables, namely `pc` and `flag`. Both state variables are n-tuples, where n refers to the number of concurrent processes. As one might expect from the name, the `pc` tuple stores the current program location pc_i for the i-th process. These possible program locations, marked in blue in Algorithm 2, are the result of applying the labeling transformation discussed in Subsection 2.1.1 and are kept in an explicitly defined set `Labels`. Similarly, the `flag` tuple represents the flag value of each process. To distinguish concurrent processes from each other, we introduced a set `P` which contains all process identifiers, ranging from 1 to n. This n is defined to be a constant in our definition and can be set during the model checking process as part of the TLC configuration. To describe the initial states of our algorithm, we defined the `Init`-predicate. Here, we require each process to start at program location "m" and with an initial flag value of 0. As stated in Subsection 2.1.1, we used the labeling transformation and the translation procedure to first label the pseudocode and then obtain the set of possible transitions. Note that we modeled missing `else` branches as `skip` statements. For each of these transitions, we created an action with a parameter `p` representing the given process identifier. We consider the action `m(p)` as an example:

$$
\begin{aligned}
&m(p) \triangleq \\
&\qquad \land\, pc[p] = \text{“m”} \\
&\qquad \land\, pc' = [pc \text{ EXCEPT } ![p] = \text{“l0”}] \\
&\qquad \land\, \mathit{flag}' = [\mathit{flag} \text{ EXCEPT } ![p] = 1]
\end{aligned}
$$

This action describes the transition from a state s with `pc[p] = "m"` to a "new" state t with `pc[p] = "l0"` and `flag[p] = 1`, where `p` is the process identifier of the current process and $s, t \in \text{St}$. As all state variables, namely `pc` and `flag`, are changed by this transition, there is no need for the *unchanged*-action.

All these actions representing state transitions are combined into a single action called `FlagWithInteger(p)` to keep the specification readable. Note that in the last action, namely `l10(p)`, we change the `pc` variable to "`m`" rather than "`m'`". We do this to save on one additional action that would only change the `pc` from "`m'`" to "`m`". The `Next`-action describes the concurrent nature of our algorithm, as discussed in Subsection 2.1.1. The specification itself is given in the standard form with a temporal formula that expresses a set of fairness constraints.

$$\begin{aligned} \mathit{Spec} \triangleq\ & \land \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \\ & \land \forall p \in P : \mathrm{WF}_{\mathit{vars}}((\mathit{pc}[p] \neq \text{“m”}) \land \mathit{FlagWithInteger}(p)) \end{aligned}$$

These weak fairness constraints ensure that every process eventually makes progress if it has shown its intention to enter the critical section and is able to make a step. Hence, we exclude the possibility that the scheduler ignores a certain process even though it would be able to progress.
To simplify the properties we want to check on our algorithm, we introduced two predicates: `insideCrit(p)` and `hasIntention(p)`. The semantics of these predicates are given in Subsection 2.1.2. In addition to mutual exclusion and starvation freedom, we also included the `TypeInvariant` property. This property ensures that the state variables and constants of the specification only take valid values. The entire specification, provided in Section A.1, totals 182 lines of code (LoC).

2.2.2 Results

The results given in Table 2.3 align with the findings of [20] with respect to mutual exclusion. Regarding starvation freedom, we note that Spronck and Luttik only checked for reachability, which states that for every concurrent process the critical section is still reachable after every non-critical step. They found reachability to be satisfied by the integer variant of Szymanski's flag algorithm. Since starvation freedom requires every process to actually enter its critical section, it implies reachability. Hence, our results align here as well. Thus, this variant of Szymanski's algorithm satisfies the given properties under weak fairness constraints.

n	Mutual Exclusion	Starvation Freedom (WF)
2	✓	✓
3	✓	✓
4	✓	✓
5	✓	✓

Table 2.3: Results of Szymanski's flag algorithm

2.3 Szymanski's Flag Algorithm using Bits

In this section, we will discuss a variation of Szymanski's flag algorithm using bits to represent the flag variable. Szymanski proposed this variation in [17] as a more efficient implementation of his original algorithm. Here, the flag variable, which is originally represented as an integer, is now given by three bit variables: `intent`, `doorIn` and `doorOut`. However, the idea of the algorithm remains the same as discussed in Section 2.2.

flag	intent	doorIn	doorOut
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	1	1	1

Table 2.4: Encoding of the flag values [17]

The possible values of the original integer flag are translated in Table 2.4. Note that the meaning of the encoded numbers is equivalent to Table 2.2. One of the main advantages of this encoding is that for every possible transition of the flag variable, given in Figure 2.2, there is only a need to change one bit — except for the transition from 2 to 4 and the transition from 4 back to 0.

Algorithm 3 Szymanski's flag algorithm with bits [20]

```
           Assum.: $\forall i \in \{1, \dots, n\}.\ intent[i] = 0 \land doorIn[i] = 0 \land doorOut[i] = 0$
$(m)$       1: $intent[i] := 1$
$(l_0)$     2: await $\forall j.\ intent[j] = 0 \lor doorIn[j] = 0$
$(l_1)$     3: $doorIn[i] := 1$
$(l_2)$     4: if $\exists j.\ intent[j] = 1 \land doorIn[j] = 0$ then
$(l_4)$     5:     $intent[i] := 0$
$(l_6)$     6:     await $\exists j.\ doorOut[j] = 1$
            7: else
$(l_5)$     8:     skip
            9: end if
$(l_3)$    10: if $intent[i] = 0$ then
$(l_8)$    11:     $intent[i] := 1$
           12: else
$(l_9)$    13:     skip
           14: end if
$(l_7)$    15: $doorOut[i] := 1$
$(l_{10})$ 16: await $\forall j < i.\ doorIn[j] = 0$
$(l_{11})$ 17: critical section
$(l_{12})$ 18: await $\forall j > i.\ doorIn[j] = 0 \lor doorOut[j] = 1$
$(l_{13})$ 19: $intent[i] := 0$
$(l_{14})$ 20: $doorIn[i] := 0$
$(l_{15})$ 21: $doorOut[i] := 0$
```

2.3.1 Specification

In this section, we provide explanation for interesting constructs in our specification while leaving out details that were already discussed in Subsection 2.2.1. For the bits variant of Szymanski's algorithm, we created a user-defined module `SzymanskiFlagBits`. This module includes the following state variables: `pc`, `intent`, `doorIn` and `doorOut`. These variables are again n-tuples, where `intent`, `doorIn` and `doorOut` represent the flag variable encoded as stated in Table 2.4. All other elements are similar to the integer variant but were modified to include the new state variables and program locations of this variant. The complete specification, given in Section A.2, comprises 246 LoC, including comments to keep the specification self-explanatory.

2.3.2 Results

The obtained results for the bits variant are stated in Table 2.5. For two concurrent processes, both mutual exclusion and starvation freedom are guaranteed under weak and strong fairness constraints. When increasing the number of processes to three and more, one can see that mutual exclusion does not hold anymore. Additionally, starvation freedom is only satisfied under strong fairness. In terms of mutual exclusion, these results again align with the findings of [20]. For starvation freedom, the results differ slightly for more than two processes, since our specification includes strong fairness constraints.

n	Mutual Exclusion	Starvation Freedom (WF)	Starvation Freedom (SF)
2	✓	✓	✓
3	✗	✗	✓
4	✗	✗	✓
5	✗	✗	✓

Table 2.5: Results of Szymanski's flag algorithm using bits

We will now study the counterexample given in Figure 2.3, which illustrates the violation of mutual exclusion for three concurrent processes. Note that for more than three processes, the counterexample provided by TLC is similar to the one presented and only differs by the additional processes staying in the first program location (m). As the counterexample will demonstrate, the problem with the bits variant lies in resetting the flag variable in multiple steps.

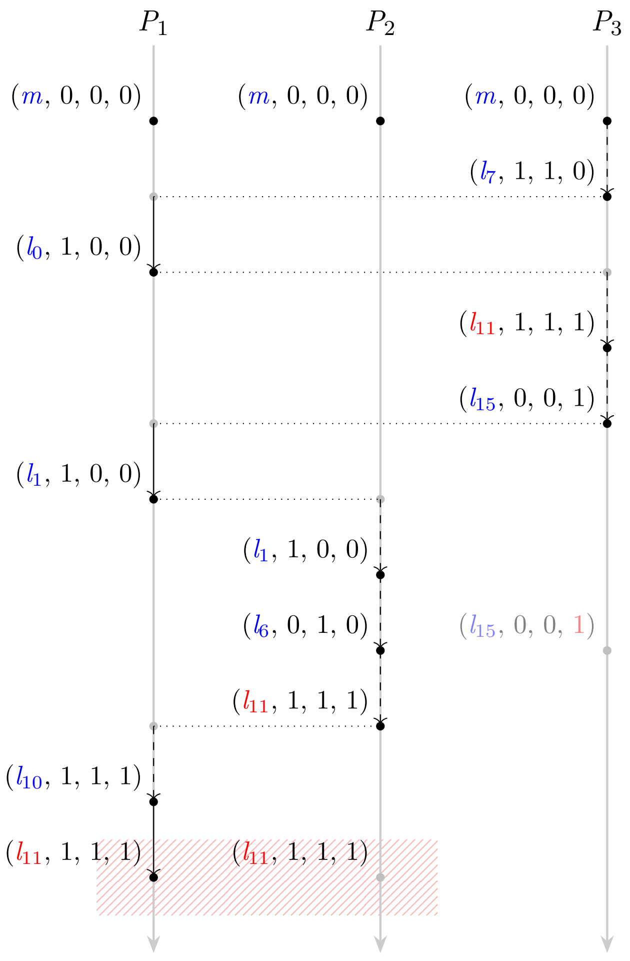

Figure 2.3: Counterexample violating mutual exclusion for $n = 3$

Before we provide a detailed explanation of the counterexample, we first discuss the semantics of the illustration. The current state of a process with process id i is given by a tuple structured as:

(`pc[i]`, `intent[i]`, `doorIn[i]`, `doorOut[i]`)

A dashed line indicates that a process progresses for more than one step, whereas a solid one indicates a single step taken. The horizontal dotted lines visualize that the scheduler gives the control to another process. Note that the index of each process represents its process identifier. Additionally, critical values of the flag variables regarding the violation of mutual exclusion are written in red.

In the beginning, all three processes P_1, P_2, P_3 are in their initial states (m). After P_3 is chosen by the scheduler, it progresses until it reaches the waiting room and passes the check on line 4, which leaves the `doorIn` open for other processes, and is now ready to pass the `doorOut` (l_7). Next, P_1 takes a step and shows its intention to enter the critical section by setting `intent` to 1 (l_0).

Following that, P_3 crosses the `doorOut` and checks if other processes with a smaller process identifier have already entered the waiting room (l_{10}). Since this is not the case, it proceeds and enters the critical section (l_{11}). After finishing its critical section, P_3 keeps progressing and starts resetting its flag variables, except for `doorOut` (l_{15}). This is a critical point, because `doorOut` $=$ 1 for any process indicates that the `doorOut` is still open for other processes to pass through (line 6). Additionally, by resetting the `doorIn` value, P_3 opens the `doorIn` as well. Hence, at this moment, both doors are open. Next, the scheduler gives control to P_1, which passes the check if the `doorIn` is open on line 2, but does not cross it yet (l_1). Afterwards, control is given to P_2, which enters the waiting room through the `doorIn` and encounters an open `doorOut` (l_6). Thus, it passes the `doorOut` and checks if any other processes with a lower process identifier are currently in the waiting room and waiting for their turn to enter the critical section. Since P_1 has not crossed the `doorIn` yet, this is not the case, and P_2 continues into its critical section (l_{11}). Now P_1 gets scheduled again and passes the `doorIn`. Since there are no other processes waiting in front of the waiting room with `intent` $= 1$ and `doorIn` $= 0$, P_1 is able to make further progress and manages to cross the `doorOut` (l_{10}). Because P_1 has the lowest process identifier of all processes (line 16), it is immediately able to enter its critical section (l_{11}). Thus, both P_1 and P_2 are in the critical section simultaneously, which violates the mutual exclusion property.

3. Conclusion

In this thesis, we modelled and verified two variants of a mutual exclusion algorithm using TLA$^+$ and compared our results to [20]. We introduced the Temporal Logic of Actions (TLA), developed by Leslie Lamport to specify and reason about concurrent systems. Next, we addressed TLA$^+$, a specification language built on top of TLA. Moreover, we gave a brief overview over model checking, the problem of state space explosion and the TLC model checker. One of the main contributions of this thesis is the presentation of a methodology applicable to similar algorithms as the ones we verified. This methodology describes a structured approach to obtaining a valid TLA$^+$ specification from the pseudocode of a concurrent algorithm, which can then be checked for the specified properties by utilizing TLC. By following this methodology, the reader is able to reproduce our results and apply the tools and techniques we discussed to similar concurrent problems. Finally, we reported on our findings about checking mutual exclusion and starvation freedom on two variants of an algorithm proposed by Szymanski in [17] to solve the mutual exclusion problem. While the claims hold for the integer variant of the algorithm, the bits variant violates the mutual exclusion property for three or more concurrent processes. The results obtained mostly align with the findings of [20], with the exception of an alternative counterexample violating mutual exclusion that we discussed in detail. Furthermore, we note that in [20] the authors only checked for reachability and not starvation freedom, since fairness constraints were outside their scope. As reachability is implied by starvation freedom, we were nonetheless able to compare the results in cases where Spronck and Luttik found reachability to be satisfied, as we checked for the stronger property under fairness constraints.

Throughout this thesis, TLA$^+$ proved to be a powerful tool that offers many useful additions through its toolbox. While TLA and TLA$^+$ have been around for many years, there seems to be increasing interest in its capabilities to reason about complex concurrent systems. This is indicated by the recent establishment of the TLA$^+$ foundation [10], reports on industrial usage through the TLA$^+$ conference [22] and publications [5, 6, 7, 8], to name a few.

References

[1] E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking, second edition.* Cyber Physical Systems Series. MIT Press, 2018. ISBN: 9780262349451.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN: 026202649X.

[3] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923. DOI: `10.1145/177492.177726`.

[4] Stephan Merz. "The Specification Language TLA+". In: *Logics of Specification Languages.* Ed. by Dines Bjørner and Martin C. Henson. Springer Berlin Heidelberg, 2008, pp. 401–451. ISBN: 978-3-540-74107-7. DOI: `10.1007/978-3-540-74107-7_8`.

[5] Chris Newcombe. "Why Amazon Chose TLA + ". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z.* Springer Berlin Heidelberg, 2014, pp. 25–39. DOI: `10.1007/978-3-662-43652-3_3`.

[6] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon Web Services Uses Formal Methods". In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: `10.1145/2699417`.

[7] Robert Beers. "Pre-RTL Formal Verification: An Intel Experience". In: Proc. DAC '08. ACM, 2008, pp. 806–811. ISBN: 9781605581156. DOI: `10.1145/1391469.1391675`.

[8] Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. "Understanding Inconsistency in Azure Cosmos DB with TLA+". In: Proc. ICSE-SEIP '23. IEEE Press, 2023, pp. 1–12. DOI: `10.1109/ICSE-SEIP58684.2023.00006`.

[9] Eric Verhulst, Raymond T. Boute, Jos Miguel Sampaio Faria, Bernhard H. C. Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS: Software Engineering for Reliable Embedded Systems.* 1st. Springer Publishing Company, 2011. ISBN: 1441997350.

[10] Noah Lehman. *Linux Foundation Announces Launch of TLA+ Foundation.* Accessed: 2024-05-01. 2023. URL: `https://www.linuxfoundation.org/press/linux-foundation-launches-tlafoundation`.

[11] Bowen Alpern and Fred B. Schneider. "Defining liveness". In: *Information Processing Letters* 21.4 (1985), pp. 181–185. DOI: `https://doi.org/10.1016/0020-0190(85)90056-0`.

[12] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2003. ISBN: 9780321143068.

[13] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. “TLA+ model checking made symbolic”. In: *Proc. OOPSLA* 3 (2019). DOI: 10.1145/3360549.

[14] Leslie Lamport. *The TLA+ Toolbox*. Accessed: 2024-05-01. 2021. URL: https://lamport.azurewebsites.net/tla/toolbox.html.

[15] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA+ Specifications”. In: *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 1999, pp. 54–66. DOI: 10.1007/3-540-48153-2_6.

[16] Leslie Lamport et al. *TLA+: A formal specification language*. https://github.com/tlaplus/tlaplus. Accessed: 2024-05-01. 2024.

[17] B. K. Szymanski. “A simple solution to Lamport’s concurrent programming problem with linear wait”. In: ICS ’88 (1988), pp. 621–626. DOI: 10.1145/55364.55425.

[18] E. W. Dijkstra. “Solution of a problem in concurrent programming control”. In: *Commun. ACM* 8.9 (1965), p. 569. DOI: 10.1145/365559.365617.

[19] Leslie Lamport. “A new solution of Dijkstra’s concurrent programming problem”. In: *Commun. ACM* 17.8 (1974), pp. 453–455. DOI: 10.1145/361082.361093.

[20] Myrthe S. C. Spronck and Bas Luttik. “Process-Algebraic Models of Multi-Writer Multi-Reader Non-Atomic Registers”. In: Proc. CONCUR’ 23 (2023). DOI: 10.4230/LIPIcs.CONCUR.2023.5.

[21] J.F. Groote and M.R. Mousavi. *Modeling and analysis of communicating systems*. The MIT Press, 2014. ISBN: 9780262027717.

[22] TLA+ Community. *TLA+ Community Event & Conference*. Accessed: 2024-07-29. 2024. URL: https://conf.tlapl.us/home/.

A. TLA$^+$ Specifications

A.1 Szymanski's Flag Algorithm

```
──────────── MODULE SzymanskiFlagInteger ────────────
EXTENDS Naturals

CONSTANTS n

VARIABLES pc, flag

Helper variables

vars ≜ ⟨pc, flag⟩

Labels ≜ {"m", "l0", "l1", "l2", "l3", "l4",
          "l5", "l6", "l7", "l8", "l9", "l10",
          "m'"}

P ≜ 1 .. n

Specification

Init ≜
    ∧ pc = [i ∈ P ↦ "m"]
    ∧ flag = [i ∈ P ↦ 0]

 flag[i] := 1
m(p) ≜
      ∧ pc[p] = "m"
      ∧ pc' = [pc EXCEPT ![p] = "l0"]
      ∧ flag' = [flag EXCEPT ![p] = 1]

 await ∀ j. flag[j] < 3
l0(p) ≜
      wait
    ∨ ( ∧ pc[p] = "l0"
        ∧ ¬(∀ j ∈ P : flag[j] < 3)
        ∧ pc' = [pc EXCEPT ![p] = "l0"]
        ∧ UNCHANGED ⟨flag⟩)
      continue
    ∨ ( ∧ pc[p] = "l0"
        ∧ ∀ j ∈ P : flag[j] < 3
        ∧ pc' = [pc EXCEPT ![p] = "l1"]
        ∧ UNCHANGED ⟨flag⟩)

 flag[i] := 3
```

```
$l1(p) \triangleq$
    $\land\ pc[p] = \text{“l1”}$
    $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l2”}]$
    $\land\ flag' = [flag \text{ EXCEPT } ![p] = 3]$

 if $\exists j.\ flag[j] = 1$
$l2(p) \triangleq$
     then
    $\lor\ (\ \land\ pc[p] = \text{“l2”}$
         $\land\ \exists j \in P : flag[j] = 1$
         $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l4”}]$
         $\land\ \text{UNCHANGED } \langle flag \rangle)$
     else
    $\lor\ (\ \land\ pc[p] = \text{“l2”}$
         $\land\ \neg(\exists j \in P : flag[j] = 1)$
         $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l5”}]$
         $\land\ \text{UNCHANGED } \langle flag \rangle)$

 then: $flag[i] := 2$
$l4(p) \triangleq$
    $\land\ pc[p] = \text{“l4”}$
    $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l6”}]$
    $\land\ flag' = [flag \text{ EXCEPT } ![p] = 2]$

 then: await $\exists j.\ flag[j] = 4$
$l6(p) \triangleq$
     wait
    $\lor\ (\ \land\ pc[p] = \text{“l6”}$
         $\land\ \neg(\exists j \in P : flag[j] = 4)$
         $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l6”}]$
         $\land\ \text{UNCHANGED } \langle flag \rangle)$
     continue
    $\lor\ (\ \land\ pc[p] = \text{“l6”}$
         $\land\ \exists j \in P : flag[j] = 4$
         $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l3”}]$
         $\land\ \text{UNCHANGED } \langle flag \rangle)$


 else: skip
$l5(p) \triangleq$
    $\land\ pc[p] = \text{“l5”}$
    $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l3”}]$
    $\land\ \text{UNCHANGED } \langle flag \rangle$

 $flag[i] := 4$
$l3(p) \triangleq$
    $\land\ pc[p] = \text{“l3”}$
    $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l7”}]$
    $\land flag' = [flag \text{ EXCEPT } ![p] = 4]$

await $\forall j < i.\ flag[j] < 2$
$l7(p) \triangleq$
    wait
    $\lor (\land pc[p] = \text{“l7”}$
        $\land \neg(\forall j \in P : (j < p) \Rightarrow (flag[j] < 2))$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l7”}]$
        $\land \text{UNCHANGED } \langle flag \rangle)$
    continue
    $\lor (\land pc[p] = \text{“l7”}$
        $\land \forall j \in P : (j < p) \Rightarrow (flag[j] < 2)$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l8”}]$
        $\land \text{UNCHANGED } \langle flag \rangle)$

critical section
$l8(p) \triangleq$
    $\land pc[p] = \text{“l8”}$
    $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l9”}]$
    $\land \text{UNCHANGED } \langle flag \rangle$

await $\forall j > i.\ flag[j] < 2 \lor flag[j] > 3$
$l9(p) \triangleq$
    wait
    $\lor (\land pc[p] = \text{“l9”}$
        $\land \neg(\forall j \in P : (j > p) \Rightarrow (flag[j] < 2 \lor flag[j] > 3))$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l9”}]$
        $\land \text{UNCHANGED } \langle flag \rangle)$
    continue
    $\lor (\land pc[p] = \text{“l9”}$
        $\land \forall j \in P : (j > p) \Rightarrow (flag[j] < 2 \lor flag[j] > 3)$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l10”}]$
        $\land \text{UNCHANGED } \langle flag \rangle)$

$flag[i] := 0$
$l10(p) \triangleq$
    $\land pc[p] = \text{“l10”}$
    $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“m”}]$  $m'$ *\
    $\land flag' = [flag \text{ EXCEPT } ![p] = 0]$

$FlagWithInteger(p) \triangleq$
    $\lor m(p)$            $flag[i] := 1$
    $\lor l0(p)$           await $\forall j.\ flag[j] < 3$
    $\lor l1(p)$           $flag[i] := 3$
    $\lor l2(p)$           if $\exists j.\ flag[i] = 1$ then
```

```
∨ $l4(p)$              $flag[i] := 2$
∨ $l6(p)$              await $\exists j.\ flag[j] = 4$
                     else
∨ $l5(p)$              skip (empty else branch)
∨ $l3(p)$            $flag[i] := 4$
∨ $l7(p)$            await $\forall j < i.\ flag[j] < 2$
∨ $l8(p)$            critical section $(crit[i]' := 1)$
∨ $l9(p)$            await $\forall j < i.\ flag[j] < 2 \lor flag[j] > 3$
∨ $l10(p)$           $flag[i] := 0\ (crit[i]' := 0)$
```

$Next \triangleq \exists p \in P : FlagWithInteger(p)$

$Spec \triangleq \land Init \land \Box[Next]_{vars}$
$\qquad\quad \land \forall p \in P : \mathrm{WF}_{vars}((pc[p] \neq \text{“m”}) \land FlagWithInteger(p))$

Predicates used for the specified properties

$insideCrit(p) \triangleq pc[p] = \text{“l8”}$

$hasIntention(p) \triangleq pc[p] = \text{“l0”}$

Properties to check

$TypeInvariant \triangleq$
$\quad \land n \in Nat$
$\quad \land pc \in [P \rightarrow Labels]$
$\quad \land flag \in [P \rightarrow 0 \,..\, 4]$

$MutualExclusion \triangleq$
$\quad \forall i, j \in P : (i \neq j) \Rightarrow \neg(insideCrit(i) \land insideCrit(j))$

$StarvationFreedom \triangleq$
$\quad \forall p \in P : \Box(hasIntention(p) \Rightarrow \Diamond(insideCrit(p)))$

A.2 Szymanski's Flag Algorithm using Bits

```
MODULE SzymanskiFlagBits

EXTENDS Naturals

CONSTANTS n

VARIABLES pc, intent, doorIn, doorOut

Helper variables

vars ≜ ⟨pc, intent, doorIn, doorOut⟩

Labels ≜ {"m", "l0", "l1", "l2", "l3", "l4", "l5", "l6", "l7",
          "l8", "l9", "l10", "l11", "l12", "l13",
          "l14", "l15", "m'",
          }

P ≜ 1 .. n

Specification

Init ≜
    ∧ pc = [i ∈ P ↦ "m"]
    ∧ intent  = [i ∈ P ↦ 0]
    ∧ doorIn = [i ∈ P ↦ 0]
    ∧ doorOut = [i ∈ P ↦ 0]

 intent[i] := 1
m(p) ≜
     ∧ pc[p] = "m"
     ∧ pc' = [pc EXCEPT ![p] = "l0"]
     ∧ intent' = [intent EXCEPT ![p] = 1]
     ∧ UNCHANGED ⟨doorIn, doorOut⟩

 await ∀j. intent[j] = 0 ∨ doorIn[j] = 0
l0(p) ≜
     wait
    ∨ ( ∧ pc[p] = "l0"
        ∧ ¬(∀ j ∈ P : intent[j] = 0 ∨ doorIn[j] = 0)
        ∧ pc' = [pc EXCEPT ![p] = "l0"]
        ∧ UNCHANGED ⟨intent, doorIn, doorOut⟩)
     continue
    ∨ ( ∧ pc[p] = "l0"
        ∧ ∀ j ∈ P : (intent[j] = 0 ∨ doorIn[j] = 0)
        ∧ pc' = [pc EXCEPT ![p] = "l1"]
        ∧ UNCHANGED ⟨intent, doorIn, doorOut⟩)
```

```
$doorIn[i] := 1$
$l1(p) \triangleq$
    $\land\ pc[p] = \text{“l1”}$
    $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l2”}]$
    $\land\ doorIn' = [doorIn \text{ EXCEPT } ![p] = 1]$
    $\land\ \text{UNCHANGED } \langle intent, doorOut \rangle$

if $\exists j.\ intent[j] = 1 \land doorIn[j] = 0$
$l2(p) \triangleq$
     then
    $\lor\ (\ \land\ pc[p] = \text{“l2”}$
        $\land\ \exists j \in P : (intent[j] = 1 \land doorIn[j] = 0)$
        $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l4”}]$
        $\land\ \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$
     else
    $\lor\ (\ \land\ pc[p] = \text{“l2”}$
        $\land\ \neg(\exists j \in P : (intent[j] = 1 \land doorIn[j] = 0))$
        $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l5”}]$
        $\land\ \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$

then: $intent[i] := 0$
$l4(p) \triangleq$
    $\land\ pc[p] = \text{“l4”}$
    $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l6”}]$
    $\land\ intent' = [intent \text{ EXCEPT } ![p] = 0]$
    $\land\ \text{UNCHANGED } \langle doorIn, doorOut \rangle$

then: await $\exists j.\ doorOut[j] = 1$
$l6(p) \triangleq$
     wait
    $\lor\ (\ \land\ pc[p] = \text{“l6”}$
        $\land\ \neg(\exists j \in P : doorOut[j] = 1)$
        $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l6”}]$
        $\land\ \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$
     continue
    $\lor\ (\ \land\ pc[p] = \text{“l6”}$
        $\land\ \exists j \in P : doorOut[j] = 1$
        $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l3”}]$
        $\land\ \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$

else: skip
$l5(p) \triangleq$
    $\land\ pc[p] = \text{“l5”}$
    $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“l3”}]$
    $\land\ \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle$
```

```
if $intent[i] = 0$
$l3(p) \triangleq$
    then
    $\lor ( \land pc[p] = \text{“l3”}$
        $\land intent[p] = 0$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l8”}]$
        $\land \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$
    else
    $\lor ( \land pc[p] = \text{“l3”}$
        $\land \neg(intent[p] = 0)$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l9”}]$
        $\land \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$

then: $intent[i] := 1$
$l8(p) \triangleq$
    $\land pc[p] = \text{“l8”}$
    $\land intent' = [intent \text{ EXCEPT } ![p] = 1]$
    $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l7”}]$
    $\land \text{UNCHANGED } \langle doorIn, doorOut \rangle$

else: skip
$l9(p) \triangleq$
    $\land pc[p] = \text{“l9”}$
    $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l7”}]$
    $\land \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle$

$doorOut[i] := 1$
$l7(p) \triangleq$
    $\land pc[p] = \text{“l7”}$
    $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l10”}]$
    $\land doorOut' = [doorOut \text{ EXCEPT } ![p] = 1]$
    $\land \text{UNCHANGED } \langle intent, doorIn \rangle$

await $\forall j < i.\ doorIn[j] = 0$
$l10(p) \triangleq$
    wait
    $\lor ( \land pc[p] = \text{“l10”}$
        $\land \neg(\forall j \in P : j < p \Rightarrow (doorIn[j] = 0))$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l10”}]$
        $\land \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$
    continue
    $\lor ( \land pc[p] = \text{“l10”}$
        $\land \forall j \in P : j < p \Rightarrow (doorIn[j] = 0)$
        $\land pc' = [pc \text{ EXCEPT } ![p] = \text{“l11”}]$
        $\land \text{UNCHANGED } \langle intent, doorIn, doorOut \rangle)$
```

```
critical section
$l11(p) \triangleq$
    $\land\ pc[p] = \text{"l11"}$
    $\land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \text{"l12"}]$
    $\land\ \text{UNCHANGED}\ \langle intent,\ doorIn,\ doorOut \rangle$

await $\forall j > i.\ doorIn[j] = 0 \lor doorOut[j] = 1$
$l12(p) \triangleq$
      wait
    $\lor\ (\ \land\ pc[p] = \text{"l12"}$
        $\land\ \neg(\forall j \in P : j > p \Rightarrow (doorIn[j] = 0 \lor doorOut[j] = 1))$
        $\land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \text{"l12"}]$
        $\land\ \text{UNCHANGED}\ \langle intent,\ doorIn,\ doorOut \rangle)$
      continue
    $\lor\ (\ \land\ pc[p] = \text{"l12"}$
        $\land\ \forall j \in P : j > p \Rightarrow (doorIn[j] = 0 \lor doorOut[j] = 1)$
        $\land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \text{"l13"}]$
        $\land\ \text{UNCHANGED}\ \langle intent,\ doorIn,\ doorOut \rangle)$

$intent[i] := 0$
$l13(p) \triangleq$
    $\land\ pc[p] = \text{"l13"}$
    $\land\ intent' = [intent\ \text{EXCEPT}\ ![p] = 0]$
    $\land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \text{"l14"}]$
    $\land\ \text{UNCHANGED}\ \langle doorIn,\ doorOut \rangle$

$doorIn[i] := 0$
$l14(p) \triangleq$
    $\land\ pc[p] = \text{"l14"}$
    $\land\ doorIn' = [doorIn\ \text{EXCEPT}\ ![p] = 0]$
    $\land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \text{"l15"}]$
    $\land\ \text{UNCHANGED}\ \langle intent,\ doorOut \rangle$

$doorOut[i] := 0\ (\ \land\ crit[i] := 0)$
$l15(p) \triangleq$
    $\land\ pc[p] = \text{"l15"}$
    $\land\ doorOut' = [doorOut\ \text{EXCEPT}\ ![p] = 0]$
    $\land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \text{"m"}]$  $m'$ *\
    $\land\ \text{UNCHANGED}\ \langle intent,\ doorIn \rangle$


$FlagWithBits(p) \triangleq$
    $\lor\ m(p)$          $intent[i] := 1$
    $\lor\ l0(p)$         await $\forall j.\ intent[j] = 0 \lor doorIn[j] = 0$
    $\lor\ l1(p)$         $doorIn[i] := 1$
    $\lor\ l2(p)$         if $\exists j.\ intent[j] = 1 \land doorIn[j] = 0$ then
```

```
$\lor\ l4(p)$                  $intent[i] := 0$
$\lor\ l6(p)$                  await $\exists j.\ doorOut[j] = 1$
                            else
$\lor\ l5(p)$                  skip (empty else branch)
$\lor\ l3(p)$                if $intent[i] = 0$ then
$\lor\ l8(p)$                  $intent[i] := 1$
                            else
$\lor\ l9(p)$                  skip (empty else branch)
$\lor\ l7(p)$                $doorOut[i] := 1$
$\lor\ l10(p)$               await $\forall j < i.\ doorIn[j] = 0$
$\lor\ l11(p)$               critical section
$\lor\ l12(p)$               await $\forall j > i.\ doorIn[j] = 0 \lor doorOut[j] = 1$
$\lor\ l13(p)$               $intent[i] := 0$
$\lor\ l14(p)$               $doorIn[i] := 0$
$\lor\ l15(p)$               $doorOut[i] := 0$
```

$$Next \triangleq \exists\, p \in P : FlagWithBits(p)$$

$$\begin{aligned} Spec \triangleq\ & \land\ Init \land \Box[Next]_{vars} \\ & \land\ \forall\, p \in P : \mathrm{WF}_{vars}((pc[p] \neq \text{“m”}) \land FlagWithBits(p)) \end{aligned}$$

Predicates used for the specified properties

$$insideCrit(p) \triangleq pc[p] = \text{“l11”}$$

$$hasIntention(p) \triangleq pc[p] = \text{“l0”}$$

Properties to check

$$\begin{aligned} & TypeInvariant \triangleq \\ & \quad \land\ n \in Nat \\ & \quad \land\ pc \in [P \to Labels] \\ & \quad \land\ intent \in [P \to \{0, 1\}] \\ & \quad \land\ doorIn \in [P \to \{0, 1\}] \\ & \quad \land\ doorOut \in [P \to \{0, 1\}] \end{aligned}$$

$$\begin{aligned} & MutualExclusion \triangleq \\ & \quad \forall\, i, j \in P : (i \neq j) \Rightarrow \neg(insideCrit(i) \land insideCrit(j)) \end{aligned}$$

$$\begin{aligned} & StarvationFreedom \triangleq \\ & \quad \forall\, p \in P : \Box(hasIntention(p) \Rightarrow \Diamond(insideCrit(p))) \end{aligned}$$

B. TLC Configurations

TLC Configuration for $n = 2$

```
CONSTANT n = 2
SPECIFICATION Spec
INVARIANT TypeInvariant
INVARIANT MutualExclusion
PROPERTY StarvationFreedom
```

TLC Configuration for $n = 3$

```
CONSTANT n = 3
SPECIFICATION Spec
INVARIANT TypeInvariant
INVARIANT MutualExclusion
PROPERTY StarvationFreedom
```

TLC Configuration for $n = 4$

```
CONSTANT n = 4
SPECIFICATION Spec
INVARIANT TypeInvariant
INVARIANT MutualExclusion
PROPERTY StarvationFreedom
```

TLC Configuration for $n = 5$

```
CONSTANT n = 5
SPECIFICATION Spec
INVARIANT TypeInvariant
INVARIANT MutualExclusion
PROPERTY StarvationFreedom
```